

# Interface Contract Enforcement for Improvement of Computational Quality of Service (CQoS) for Scientific Components

[Extended Abstract]

Li Li

Argonne National Laboratory  
Mathematics and Computer  
Science Division  
likli@mcs.anl.gov

Lois Curfman McInnes

Argonne National Laboratory  
Mathematics and Computer  
Science Division  
curfman@mcs.anl.gov

Tamara L. Dahlgren

Lawrence Livermore National  
Laboratory  
Center for Applied Scientific  
Computing  
dahlgrenl@llnl.gov

Boyana Norris

Argonne National Laboratory  
Mathematics and Computer  
Science Division  
norris@mcs.anl.gov

## ABSTRACT

This paper describes recent investigations into improving the quality and performance of component-based scientific software. Our approach merges work on Computational Quality of Service (CQoS) with enforceable semantic annotations, in the form of interface contracts, to enhance the adaptivity of component-based applications and improve the usability of CQoS components.

Component interfaces, as advanced by the Common Component Architecture (CCA) Forum, enable easy access to complex software packages for high-performance scientific computing. However, many challenges remain in ensuring that components are configured and used correctly in the context of long-running simulations. Interface contracts have proven to be helpful for ensuring correct usage. Additional work on Computational Quality of Service (CQoS) exploits component automation, including capabilities for plugging and unplugging components during execution, to help application scientists choose among alternative algorithmic implementations and parameters, thereby creating new opportunities to enhance the performance of CCA applications.

The integration of CQoS and interface contracts is described. Two application use cases involving solver components are also presented.

## Keywords

Common Component Architecture, Computational Quality of Service, Interface Contracts

## 1. INTRODUCTION

As computational science progresses toward ever more realistic multiphysics and multiscale applications, no single research group can effectively develop, select, or tune all of the components in a given application. Furthermore, no single tool, solver, or solution strategy can seamlessly span the entire spectrum efficiently. Component-based software engineering approaches help manage some of the complexity of developing such large scientific applications.

The Common Component Architecture (CCA) [1] defines a component software engineering approach specifically targeted at high-performance computing (HPC) applications. The challenge then becomes how to make sound choices from among the available implementations and parameters, with suitable tradeoffs among performance, accuracy, mathematical consistency, and reliability, both when initially composing and configuring a component application, and when dynamically adapting to respond to continuous changes in component requirements and execution environments.

To at least partially automate the process of characterizing the performance of component applications and selecting and configuring particular implementations, we have introduced the concept of computational quality of service (CQoS) [9], or the automatic composition, substitution, and dynamic reconfiguration of components to suit a particular computational purpose and environment. CQoS embodies the familiar concept of quality of service in networking as well as the ability to specify and manage characteristics of the application in a way that adapts to the changing (computational) environment. The two main facets of CQoS tools, therefore, are measurement and analysis infrastructure and control infrastructure for dynamic component replacement and domain-specific decision making. This paper focuses on

the performance and metadata management and analysis support provided by the CQoS infrastructure.

Because scientific components are developed by people with different backgrounds and training, it is not safe to assume that everyone uses the same level of rigor in their software development practices - especially in the case of research software. Interface contract enforcement [4,5] is intended to help scientists gain confidence in software built from components. Hence, executable interface contracts provide some assurances that interface failures can be caught regardless of the programming discipline used by component implementors.

In this paper, we focus on interface contract enforcement within the context of CQoS. We present different types of CQoS-contracts and their uses in two parallel application contexts.

## 2. CONTRACT ENFORCEMENT IN CQOS

CQoS expands on traditional QoS ideas by considering application-specific metrics, or metadata, which enable the annotation and characterization of component performance. Before automating the selection of component instances, however, one must be able to collect and analyze performance information and related metadata. The CQoS database interface [8] has been designed to support the management and analysis of performance and application metadata, so the mapping of a problem to an implementation with the potential of yielding the best performance can be accomplished statically or at runtime. There are two types of components for storing and querying CQoS performance data and metadata. The database component provides general-purpose interfaces for storing and accessing data in a physical database. Comparator interfaces compare and/or match properties of two problems under user-specified conditions. We incorporated PerfExplorer [6] into CQoS analysis infrastructure to support performance analysis and decision-making for runtime adaptivity. PerfExplorer supports the construction of a runtime parameter recommendation system, using classification capabilities in the Weka [12] data-mining package.

With database and analysis support, our CQoS approach to automating parallel application configuration involves three phases: collection of performance data in a training database, performance analysis, and adaptive application composition. The training data for analysis is generated by executing the application multiple times, varying key parameters that have an effect on the total runtime. After the performance and associated metadata are stored and queried through the general CQoS database component interfaces, PerfExplorer loads the data and builds a classifier from it. For production application runs, the classifier is loaded into a CCA component. A appropriate parameter setting is obtained by querying the classifier with the current values of application-specific metadata. These values are matched to the classification properties to find the best class selection for the parameters. Furthermore, our comparator components serve as filters when searching for appropriate parameter settings in the database. A detailed description of our CQoS approach to runtime adaption is presented in [7].

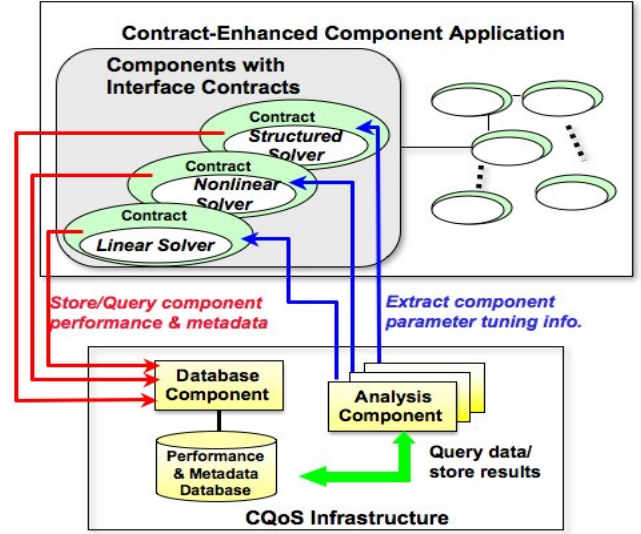


Figure 1: A contract-enhanced CQoS component application

Enforceable interface contracts provide a uniform mechanism for documenting behaviors and constraints associated with components, automatically verifying component implementation conformance to the documented specification, and checking that components are used by applications in a manner consistent with their design specifications. Support for semantic annotations has been integrated into component interface definition, thereby extending basic method signatures for the expression of interface contracts independent of the implementation languages of the associated components. Currently, the specification of *pre-* and *post-conditions* is supported and tested for some languages. *Pre-* and *post-conditions* specify conditions that must be satisfied prior to and immediately after invocation of component interfaces, respectively. In the context of CQoS, the contracts can be enforced to validate component arguments, evaluate component results, and invoke CQoS database and analysis functions in an integrated manner without modifying component internals.

### 2.1 Integrate Interface Contracts and CQoS

Figure 1 illustrates how the interface contracts are enforced and integrated into CQoS infrastructure. In an adaptive component application, the generic component interfaces are extended with semantic contracts that specify the behavioral constraints of using the components. Beside checking the validity of component use, the contracts can be helpful to support adaptive runtime decision-making. The contracts can store component performance and meta-data, as shown in red arrows in the figure, and query for tuning suggestion, in blue arrows, through CQoS database and analysis functions. Easy control (turned off or on at runtime, e.g., for performance reasons) of contracts makes it easier to perform adaptivity-related checks. Moreover, encapsulating such checks in individual component specifications facilitates the component-specific adaptive decision-making.

### 2.2 CQoS Interface Contract Types

CQoS interface contract types falls into two categories according to their functions.

- *Argument validator.* The contracts, mostly precondition clauses, ensure that appropriate component arguments and parameters are used. They raise violation exceptions when the conditions are not met.
- *Performance data collector and evaluator.* The contracts collect and evaluate a component's performance in post-conditions. Postcondition violations are raised as exceptions when the component performance is problematic.

To handle contract-raised exceptions in the case of pre- and post-condition violations and to perform CQoS-related tasks, two classes of functions are needed:

- *Proxy that handles external calls to CQoS functions.* The functions invoke database and analysis capabilities in CQoS, e.g., call database functions to store and manage performance and meta-data, and query a Perf-Explorer classifier for parameter tuning information.
- *Adaptive decision maker.* Based on tuning suggestions returned from CQoS analysis facilities or application-specific adaptation heuristics, the exception handlers make adaptation decision at runtime. The functions are invoked in response to problematic performance of a component execution to adjust component parameters, reconfigure, substitute, or recompose components to be used in subsequent iterations of program run.

### 3. APPLICATION USE CASES

Interface contracts, described in Section 2, have been employed in two different application contexts.

**Towards Optimal Petascale Simulations (TOPS).** TOPS [3, 10] solver components provide high-level access to a parallel (non-)linear algebraic solvers for large linear and nonlinear algebraic systems arising from either structured or unstructured meshes. The common interfaces employed by TOPS solver components enable easy access to suites of independently developed algorithms and implementations. The challenge then becomes how, during runtime, to make the best choices for reliability, accuracy, and performance. We have been extending TOPS components to incorporate new CQoS capabilities to facilitate appropriate choices for algorithms and parameters of TOPS linear and nonlinear solver components. A important task is to incorporate interface contract enforcement capabilities into the TOPS component specification to improve performance and robustness. First of all, contracts are enforced to ensure suitable TOPS component algorithms and parameters for nonlinear PDE applications. Second, contracts perform the performance check and evaluation as needed by CQoS analysis and trigger relevant adaptivity action(s) by using CQoS functions. Figure 2 shows a TOPS functions, *computeResidual* that uses pre- and post- conditions to check validity of arguments and result.

```
int computeResidual(in array<double> x,
                   in array<double> f)

throws
    sidl.PreViolation, sidl.PostViolation;
require
    not_null_x : x != null;
    not_null_f : f != null;
    x_is_2d : dimen(x) == 2;
    f_is_2d : dimen(f) == 2;
ensure
    non_negative_result : result > 0;
```

Figure 2: An example TOPS component function that uses pre- and post-condition checks.

**Flow in a Driven Cavity.** Driven cavity flow combines lid-driven flow and buoyancy-driven flow in a two-dimensional rectangular cavity. We use a velocity-vorticity formulation of the Navier-Stokes and energy equations, which we discretize using a standard finite-difference scheme with a five-point stencil for each component on a uniform Cartesian mesh; see [2] for a detailed problem description. Driven cavity is an important parallel, nonlinear partial differential equation (PDE) application in the CQoS testbed. We use the contracts described in Section 2 to implement adaptive linear solver components for the application. The contracts evaluate linear solver performance in a post-condition (whether they fail or succeed using the solver), as displayed in top right box in Figure 3. An exception is raised to handle solver failure and to change the linear solver to be used for subsequent iterations. The exception handler is shown in bottom right code segment in Figure 3.

### 4. CONCLUSION

Behavioral semantic contracts have been added to scientific component specifications for validating component parameters and improving performance and robustness. These contract-enhanced components provide easy interfaces to CQoS infrastructure, which has the goal of enabling automated component selection and (re-)configuration of component-based scientific applications.

### 5. ACKNOWLEDGMENTS

We thank all members of the CCA Forum for stimulating discussions on high-performance scientific software. This work was supported in part by the Office of Advanced Scientific Computing Research via the Scientific Discovery through Advanced Computing (SciDAC) initiative [11], Office of Science, U.S. Department of Energy, under Contracts DE-AC02-06CH11357 and DE-AC52-07NA27344.

### 6. REFERENCES

- [1] D. E. Bernholdt, B. A. Allan, R. Armstrong, F. Bertrand, K. Chiu, T. L. Dahlgren, K. Damevski, W. R. Elwasif, T. G. W. Epperly, M. Govindaraju, D. S. Katz, J. A. Kohl, M. Krishnan, G. Kumfert, J. W. Larson, S. Lefantzi, M. J. Lewis, A. D. Malony,

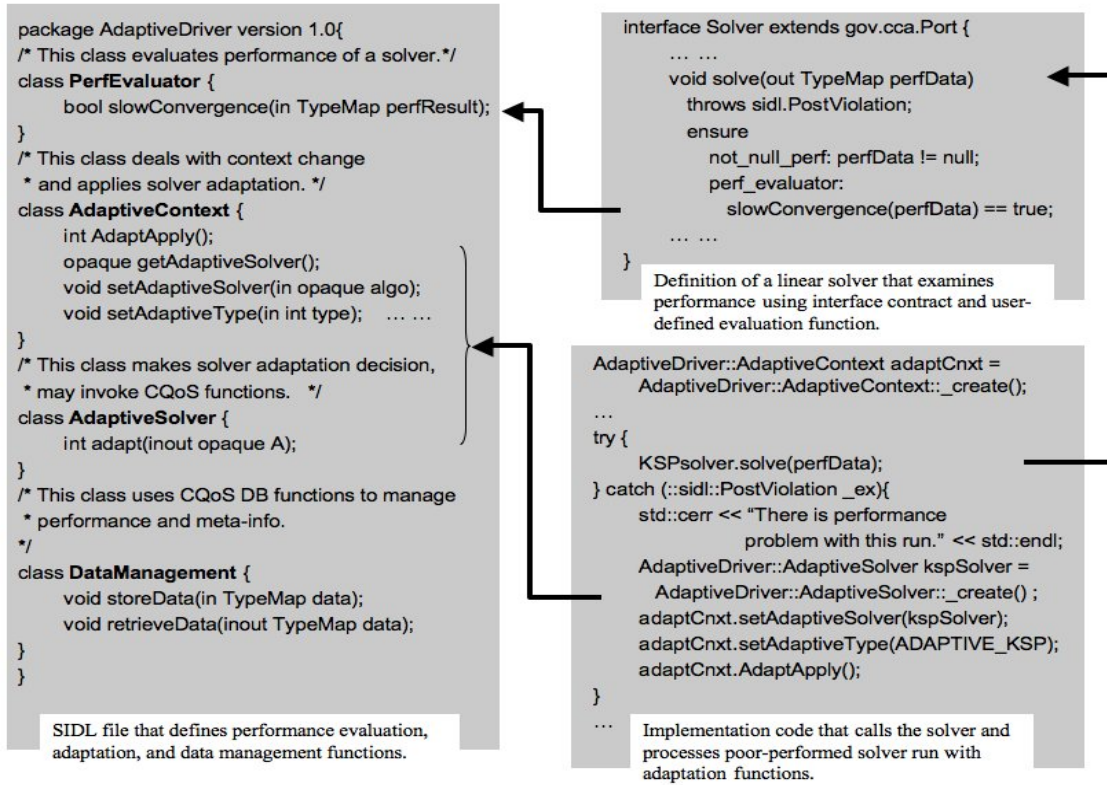


Figure 3: Interface specification and implementation in adaptive Driven Cavity that involve contract use and exception handling.

- [1] L. C. McInnes, J. Nieplocha, B. Norris, S. G. Parker, J. Ray, S. Shende, T. L. Windus, and S. Zhou. A component architecture for high-performance scientific computing. *International Journal of High-Performance Computing Applications*, pages 215–229, 2006.
- [2] T. Coffey, C. Kelley, and D. Keyes. Pseudo-transient continuation and differential algebraic equations. *SIAM J. Sci. Comp.*, 25:553–569, 2003.
- [3] D. Keyes (PI). Terascale Optimal PDE Simulations (TOPS) Center. <http://tops-scidac.org/>, 2006.
- [4] T. L. Dahlgren. *Performance-Driven Interface Contract Enforcement for Scientific Components*. PhD thesis, University of California, Davis, One Shields Avenue, Davis, CA, 95616, 2008.
- [5] T. L. Dahlgren and P. T. Devanbu. Improving scientific software component quality through assertions. In *Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications*, pages 73–77, St. Louis, Missouri, May 2005.
- [6] K. A. Huck, A. D. Malony, S. Shende, and A. Morris. Scalable, automated performance analysis with tau and perexplorer. In *Parallel Computing (ParCo)*, Aachen, Germany, 2007.
- [7] L. Li, J. P. Kenny, M. Wu, K. Huck, and et al. Adaptive application composition in quantum chemistry. In *Proceedings of the 5th International Conference on the Quality of Software Architectures (QoSA 2009)*, 2009.
- [8] L. Li, B. Norris, H. Johansson, L. McInnes, and J. Ray. Component infrastructure for managing performance data and runtime adaptation of parallel applications. In *Proceedings of PARA08 (9th International Workshop on State-of-the-Art in Scientific and Parallel Computing)*, 2008.
- [9] B. Norris, J. Ray, R. Armstrong, L. C. McInnes, D. E. Bernholdt, W. R. Elwasif, A. D. Malony, and S. Shende. Computational quality of service for scientific components. In *Proceedings of International Symposium on Component-Based Software Engineering (CBSE7)*, Edinburgh, Scotland, 2004.
- [10] B. Smith et al. TOPS Solver Components. <http://www.mcs.anl.gov/scidac-tops/solver-components/tops.html>, 2005.
- [11] U. S. Dept. of Energy. SciDAC Initiative homepage. <http://www.osti.gov/scidac/>, 2006.
- [12] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, Second edition, 2005.

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.